

Object Oriented
Programming
in Java



»Лекционен курс

»ООП1 (Java)



Наследяване (2) >



Регистрация

<https://tinyurl.com/2aycbz87>



Кратко повторение

Синтаксис на наследяване

- » Наследяването е неотменна част от Java (и всички ОО езици изобщо).
- » По принцип винаги извършваме наследяване, когато създаваме клас, дори явно да не наследяваме друг клас.
 - > Безусловно наследяваме стандартния базов клас на Java, наречен **Object**.
- » Синтаксисът на наследяването използва съвсем различна форма от композицията.
 - > Идеята е да покажем, че „Един нов клас е като някой съществуващ клас“.
 - > Ключова дума **extends**.
 - > Новият клас автоматично получава всички данни и методи на базовия клас.

Пример



Какво прави програмата?

```
class Cleanser {  
    private String s = new String("Cleanser");  
    public void append(String a) { s += a; }  
    public void dilute() { append(" dilute()"); }  
    public void apply() { append(" apply()"); }  
    public void scrub() { append(" scrub()"); }  
    public void print() { System.out.println(s); }  
  
    public static void main(String[] args) {  
        Cleanser x = new Cleanser();  
        x.dilute(); x.apply(); x.scrub();  
        x.print();  
    }  
}
```

Пример

Коментар

```
public class Detergent extends Cleanser {  
    public void scrub() { ← препокрива метод  
        append(" Detergent.scrub() ");  
        super.scrub(); ← извикване версията  
    } ← от базовия клас  
  
    public void foam() { ← добавяне нов метод  
        append(" foam() "); ← към интерфейса  
    }  
  
    public static void main(String[ ] args) { ← тестване на новия  
        Detergent x = new Detergent(); ← клас  
        x.dilute();  
        x.apply();  
        x.scrub();  
        x.foam();  
        x.print();  
        System.out.println("Testing base class:");  
        Cleanser.main(args); ← извикване main метод на  
    } ← базовия клас
```

```
class Cleanser {  
    private String s = new String("Cleanser");  
    public void append(String a) { s += a; }  
    public void dilute() { append(" dilute()"); }  
    public void apply() { append(" apply()"); }  
    public void scrub() { append(" scrub()"); }  
    public void print() { System.out.println(s); }  
    public static void main(String[] args) {  
        Cleanser x = new Cleanser();  
        x.dilute(); x.apply(); x.scrub();  
        x.print();  
    }  
}
```

```
public class Detergent extends Cleanser {  
    public void scrub() {  
        append(" Detergent.scrub()");  
        super.scrub();  
    }  
    public void foam() {  
        append(" foam()");  
    }  
    public static void main(String[ ] args) {  
        Detergent x = new Detergent();  
        x.dilute();  
        x.apply();  
        x.scrub();  
        x.foam();  
        x.print();  
        System.out.println("Testing base class:");  
        Cleanser.main(args);  
    }  
}
```



Какъв резултат?

```
class Cleanser {  
    private String s = new String("Cleanser");  
    public void append(String a) { s += a; }  
    public void dilute() { append(" dilute()"); }  
    public void apply() { append(" apply()"); }  
    public void scrub() { append(" scrub()"); }  
    public void print() { System.out.println(s); }  
    public static void main(String[] args) {  
        Cleanser x = new Cleanser();  
        x.dilute(); x.apply(); x.scrub();  
        x.print();  
    }  
}
```



Какъв резултат?

```
Cleanser dilute() apply() scrub()  
  
Process finished with exit code 0
```

```
public class Detergent extends Cleanser {  
    public void scrub() {  
        append(" Detergent.scrub()");  
        super.scrub();  
    }  
    public void foam() {  
        append(" foam()");  
    }  
    public static void main(String[ ] args) {  
        Detergent x = new Detergent();  
        x.dilute();  
        x.apply();  
        x.scrub();  
        x.foam();  
        x.print();  
        System.out.println("Testing base class:");  
        Cleanser.main(args);  
    }  
}
```

```
Cleanser dilute() apply() Detergent.scrub() scrub() foam()  
Testing base class:  
Cleanser dilute() apply() scrub()  
  
Process finished with exit code 0
```

```

class Cleanser {
    private String s = new String("Cleanser");
    public void append(String a) { s += a; }
    public void dilute() { append(" dilute()"); }
    public void apply() { append(" apply()"); }
    public void scrub() { append(" scrub()"); }
    public void print() { System.out.println(s); }
    public static void main(String[] args) {
        Cleanser x = new Cleanser();
        x.dilute(); x.apply(); x.scrub();
        x.print();
    }
}

public class Detergent extends Cleanser {
    public void scrub() {
        append(" Detergent.scrub()");
        super.scrub();
    }
    public void foam() {
        append(" foam()");
    }
    public static void main(String[ ] args) {
        Detergent x = new Detergent();
        x.dilute();
        x.apply();
        x.scrub();
        x.foam();
        x.print();
        System.out.println("Testing base class:");
        Cleanser.main(args);
    }
}

```

- Двата класа съдържат **main** методи – могат независимо един от друг да бъдат извикани от командния ред.
- Може да се задават за всеки клас.
- Такъв стил се препоръчва – тестовият код е обвит от клас
- Когато приключим с тестването, не е необходимо да изтриваме **main** – можем да го оставим за бъдещо тестване.

```
class Cleanser {  
    private String s = new String("Cleanser");  
    public void append(String a) { s += a; }  
    public void dilute() { append(" dilute()"); }  
    public void apply() { append(" apply()"); }  
    public void scrub() { append(" scrub()"); }  
    public void print() { System.out.println(s); }  
    public static void main(String[] args) {  
        Cleanser x = new Cleanser();  
        x.dilute(); x.apply(); x.scrub();  
        x.print();  
    }  
}
```

```
public class Detergent extends Cleanser {  
    public void scrub() {  
        append(" Detergent.scrub()");  
        super.scrub();  
    }  
    public void foam() {  
        append(" foam()");  
    }  
    public static void main(String[ ] args) {  
        Detergent x = new Detergent();  
        x.dilute();  
        x.apply();  
        x.scrub();  
        x.foam();  
        x.print();  
        System.out.println("Testing base class:");  
        Cleanser.main(args);  
    }  
}
```

- Detergent.main явно извиква Cleanser.main.
- Препредава му същите аргументи от околната среда (командния ред).
- Възможно е също да се предава произволен масив от низове.

```

class Cleanser {
    private String s = new String("Cleanser");
    public void append(String a) { s += a; }
    public void dilute() { append(" dilute()"); }
    public void apply() { append(" apply()"); }
    public void scrub() { append(" scrub()"); }
    public void print() { System.out.println(s); }
    public static void main(String[] args) {
        Cleanser x = new Cleanser();
        x.dilute(); x.apply(); x.scrub();
        x.print();
    }
}

public class Detergent extends Cleanser {
    public void scrub() {
        append(" Detergent.scrub()");
        super.scrub();
    }
    public void foam() {
        append(" foam()");
    }
    public static void main(String[ ] args) {
        Detergent x = new Detergent();
        x.dilute();
        x.apply();
        x.scrub();
        x.foam();
        x.print();
        System.out.println("Testing base class:");
        Cleanser.main(args);
    }
}

```

- Особено важно! Всички методи в Cleanser са **public**.
- Ако клас от **друг пакет** е наследник на Cleanser, той има **право** на достъп само до public членовете.
- Detergent няма проблеми.

Основно правило при наследяване:

- Всички полета **private**
- Всички методи **public**

```

class Cleanser {
    private String s = new String("Cleanser");
    public void append(String a) { s += a; }
    public void dilute() { append(" dilute()"); }
    public void apply() { append(" apply()"); }
    public void scrub() { append(" scrub()"); }
    public void print() { System.out.println(s); }
    public static void main(String[] args) {
        Cleanser x = new Cleanser();
        x.dilute(); x.apply(); x.scrub();
        x.print();
    }
}

public class Detergent extends Cleanser {
    public void scrub() {
        append(" Detergent.scrub()");
        super.scrub();
    }
    public void foam() {
        append(" foam()");
    }
    public static void main(String[ ] args) {
        Detergent x = new Detergent();
        x.dilute();
        x.apply();
        x.scrub();
        x.foam();
        x.print();
        System.out.println("Testing base class:");
        Cleanser.main(args);
    }
}

```

- Тези методи са от **интерфейса** на Cleanser.
- Detergent (понеже е произведен на Cleanser) **автоматично** получава тези методи в своя интерфейс (дори и да не са явно дефинирани в Detergent).

Следователно, можем да мислим за наследяването като за повторно използване на интерфейса.

```

class Cleanser {
    private String s = new String("Cleanser");
    public void append(String a) { s += a; }
    public void dilute() { append(" dilute()"); }
    public void apply() { append(" apply()"); }
    public void scrub() { append(" scrub()"); }
    public void print() { System.out.println(s); }
    public static void main(String[] args) {
        Cleanser x = new Cleanser();
        x.dilute(); x.apply(); x.scrub();
        x.print();
    }
}

```

```

public class Detergent extends Cleanser {
    public void scrub() {
        append(" Detergent.scrub()");
        super.scrub();
    }
    public void foam() {
        append(" foam()");
    }
    public static void main(String[ ] args) {
        Detergent x = new Detergent();
        x.dilute();
        x.apply();
        x.scrub();
        x.foam();
        x.print();
        System.out.println("Testing base class:");
        Cleanser.main(args);
    }
}

```



Заш?

- Възможно е да променим метод, използван в базовия клас (подобно на scrub()).
- В този случай може да искаме да извикаме **метода на базовия клас** в рамките на новата версия на метода - използваме ключовата дума **super**, която се отнася за „супер класа“, от който е наследен дадения клас.
- В scrub() **не можем** просто да извикаме scrub().

```
class Cleanser {  
    private String s = new String("Cleanser");  
    public void append(String a) { s += a; }  
    public void dilute() { append(" dilute()"); }  
    public void apply() { append(" apply()"); }  
    public void scrub() { append(" scrub()"); }  
    public void print() { System.out.println(s); }  
    public static void main(String[] args) {  
        Cleanser x = new Cleanser();  
        x.dilute(); x.apply(); x.scrub();  
        x.print();  
    }  
}
```

```
public class Detergent extends Cleanser {  
    public void scrub() {  
        append(" Detergent.scrub()");  
        super.scrub();  
    }  
    public void foam() {  
        append(" foam()");  
    }  
    public static void main(String[ ] args) {  
        Detergent x = new Detergent();  
        x.dilute();  
        x.apply();  
        x.scrub();  
        x.foam();  
        x.print();  
        System.out.println("Testing base class:");  
        Cleanser.main(args);  
    }  
}
```



Зад?

Ще предизвика нежелана
рекурсия



```
class Cleanser {  
    private String s = new String("Cleanser");  
    public void append(String a) { s += a; }  
    public void dilute() { append(" dilute()"); }  
    public void apply() { append(" apply()"); }  
    public void scrub() { append(" scrub()"); }  
    public void print() { System.out.println(s); }  
    public static void main(String[] args) {  
        Cleanser x = new Cleanser();  
        x.dilute(); x.apply(); x.scrub();  
        x.print();  
    }  
}
```

```
public class Detergent extends Cleanser {  
    public void scrub() {  
        append(" Detergent.scrub()");  
        super.scrub();  
    }  
    public void foam() {  
        append(" foam()");  
    }  
    public static void main(String[ ] args) {  
        Detergent x = new Detergent();  
        x.dilute();  
        x.apply();  
        x.scrub();  
        x.foam();  
        x.print();  
        System.out.println("Testing base class:");  
        Cleanser.main(args);  
    }  
}
```



- Когато наследяваме не сме ограничени да използваме само методите на базовия клас .
- Можем да добавяме нови методи, напр. `foam()`.

Инициализация на базовите класове



16

Инициализация на базови класове

- » При наследяването различаваме **два класа**
 - > Базов;
 - > Произведен.
- » Наследяването **не е просто копиране** на интерфейса на базовия клас.
 - > Когато създаваме обект на произведен клас, той съдържа в себе си **подобект на базовия клас**.
 - > Този подобект е като, че сме създали **обект на самия базов клас**.
 - > Обектът на базовия клас е обвит от обекта на производния клас.

Инициализация на базови класове

- » От съществено значение е подобектът на базовия клас да бъде **коректно инициализиран**.
 - > Само един гарантиращ това начин - инициализация в **конструктора**, като се извиква конструктора на базовия клас.
 - > Java **автоматично** вмъква в конструктора на производния клас извикване на конструктора на базовия клас.

Инициализация на базов клас

- » От съществено значение е подобектът на базовия клас да бъде **коректно инициализиран**.
- » Само един начин, гарантиращ това ...



Кой?

Инициализация на базов клас

- » От съществено значение е подобектът на базовия клас да бъде **коректно инициализиран**
- » Само един начин, гарантиращ това ...



Извикване **конструктора на базовия клас:**

- Има необходимите **права** за извършване инициализация на базовия клас.
- Java **автоматично** вмъква извиквания към конструктора по подразбиране на базовия клас в конструктора на производния клас.

Пример



Результат?

```
class Plant {  
    Plant() {  
        System.out.println("Plant constructor");  
    }  
}  
  
class Tree extends Plant {  
    Tree() {  
        System.out.println("Tree constructor");  
    }  
}  
  
public class Elm extends Tree {  
    Elm() {  
        System.out.println("Elm constructor");  
    }  
}  
  
public static void main(String[ ] args) {  
    Elm e = new Elm();  
}
```



Пример



Результат?

```
class Plant {  
    Plant() {  
        System.out.println("Plant constructor");  
    }  
}  
  
class Tree extends Plant {  
    Tree() {  
        System.out.println("Tree constructor");  
    }  
}  
  
public class Elm extends Tree {  
    Elm() {  
        System.out.println("Elm constructor");  
    }  
  
    public static void main(String[ ] args) {  
        Elm e = new Elm();  
    }  
}
```



Plant constructor
Tree constructor
Elm constructor

Process finished with exit code 0

Пример

```
class Plant {  
    Plant() {  
        System.out.println("Plant constructor");  
    }  
  
    class Tree extends Plant {  
        Tree() {  
            System.out.println("Tree constructor");  
        }  
    }  
  
    public class Elm extends Tree {  
        Elm() {  
            System.out.println("Elm constructor");  
        }  
  
        public static void main(String[ ] args) {  
            Elm e = new Elm();  
        }  
    }  
}
```



Действието на конструктора е в посока от базовия клас „навън“, така че **базовият клас се инициализира преди конструкторите на производния клас да могат да осъществяват достъп до него.**

Пример

```
class Plant {  
    Plant() {  
        System.out.println("Plant constructor");  
    }  
  
    class Tree extends Plant {  
        Tree() {  
            System.out.println("Tree constructor");  
        }  
    }  
  
    public class Elm extends Tree {  
        Elm() {  
            System.out.println("Elm constructor");  
        }  
  
        public static void main(String[ ] args) {  
            Elm e = new Elm();  
        }  
    }  
}
```



Дори да не създадем конструктор за Elm(), компилаторът ще синтезира вместо нас тъкъв по подразбиране, който ще извика конструктора на базовия клас.

Пример: Chess

```
class Game {  
    Game(int i) {  
        System.out.println("Game constructor");  
    }  
}  
class BordGame extends Game {  
    BordGame(int i) {  
        System.out.println("BordGame constructor");  
    }  
}  
public class Chess extends BordGame {  
    Chess() {  
        System.out.println("Chess constructor");  
    }  
  
    public static void main(String[ ] args) {  
        Chess x = new Chess();  
    }  
}
```



Разлика с Ем?

Конструкторът на базовия
клас е с параметри.

Пример: Chess

```
class Game {  
    Game(int i) {  
        System.out.println("Game constructor");  
    }  
}  
  
class BordGame extends Game {  
    BordGame(int i) {  
  
        System.out.println("BordGame constructor");  
    }  
}  
  
public class Chess extends BordGame {  
    Chess() {  
  
        System.out.println("Chess constructor");  
    }  
  
    public static void main(String[ ] args) {  
        Chess x = new Chess();  
    }  
}
```



Как ще бъдат предадени аргументите?

Пример: Chess

```
class Game {  
    Game(int i) {  
        System.out.println("Game constructor");  
    }  
}  
  
class BordGame extends Game {  
    BordGame(int i) {  
        super(i);  
        System.out.println("BordGame constructor");  
    }  
}  
  
public class Chess extends BordGame {  
    Chess() {  
        super(11);  
        System.out.println("Chess constructor");  
    }  
  
    public static void main(String[ ] args) {  
        Chess x = new Chess();  
    }  
}
```



Как ще бъдат предадени аргументите?

super

Пример: Chess



Результат?

```
class Game {  
    Game(int i) {  
        System.out.println("Game constructor");  
    }  
}  
  
class BordGame extends Game {  
    BordGame(int i) {  
        super(i);  
        System.out.println("BordGame constructor");  
    }  
}  
  
public class Chess extends BordGame {  
    Chess() {  
        super(11);  
        System.out.println("Chess constructor");  
    }  
  
    public static void main(String[ ] args) {  
        Chess x = new Chess();  
    }  
}
```

Game constructor

BordGame constructor

Chess constructor

Process finished with exit code 0

Конструктори с аргументи

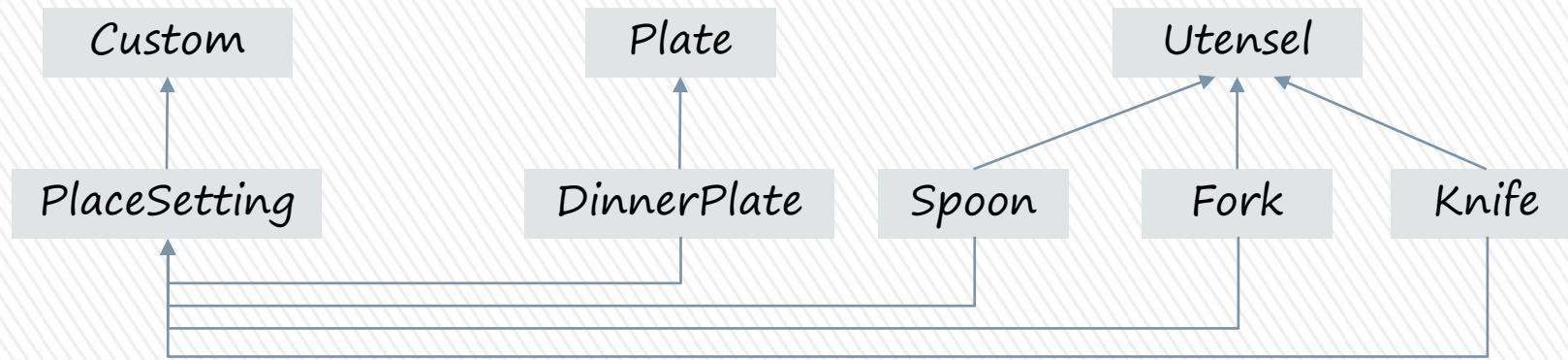
- » В предишния пример се използват **конструктори по подразбиране**.
 - > Без аргументи.
 - > Не създават проблеми на компилатора.
- » Ако искаме да използваме конструктор на базов клас **C аргументи** е необходима ключовата дума **super**.
 - > Освен това, извикването на конструктора на базовия клас трява да бъде първият оператор в конструктора на производния клас.
 - > Ако не сме го направили компилаторът ще ни напомни.

Композиция и наследяване

Комбиниране композиция и наследяване

- » В много случаи е **целесъобразно** композицията и наследяването да се използват едновременно.
- » В тези случаи е целесъобразно създаване на **посложни класове** като се използват наследяване и композиция.
 - > Заедно с необходимата инициализация с помощта на конструктори.

Пример



Пример

```
class Plate {
    Plate(int i) {
        System.out.println("Plate constructor");
    }
}
class DinnerPlate extends Plate {
    DinnerPlate(int i) {
        super(i);
        System.out.println("DinnerPlate constructor");
    }
}
class Utensil {
    Utensil(int i) {
        System.out.println("Utensil constructor");
    }
}
class Spoon extends Utensil {
    Spoon(int i) {
        super(i);
        System.out.println("Spoon constructor");
    }
}
class Fork extends Utensil {
    Fork(int i) {
        super(i);
        System.out.println("Fork constructor");
    }
}
```

Пример

```
class Knife extends Utensil {
    Knife(int i) {
        super(i);
        System.out.println("Knife constructor");
    }
}
class Custom {
    Custom(int i) {
        System.out.println("Custom constructor");
    }
}
public class PlaceSetting extends Custom {
    Spoon sp;      Fork frk;
    Knife kn;      DinnerPlate pl;
    PlaceSetting(int i) {
        super(i + 1);
        sp = new Spoon(i + 2);
        frk = new Fork(i + 3);
        kn = new Knife(i + 4);
        pl = new DinnerPlate(i + 5);
        System.out.println("PlaceSetting constructor");
    }
    public static void main(String[ ] args) {
        PlaceSetting x = new PlaceSetting(9);
    }
}
```

Пример

```
class Knife extends Utensil {  
    Knife(int i) {  
        super(i);  
        System.out.println("Knife constructor");  
    }  
}  
class Custom {  
    Custom(int i) {  
        System.out.println("Custom constructor");  
    }  
}  
public class PlaceSetting extends Custom {  
    Spoon sp;      Fork frk;  
    Knife kn;      DinnerPlate pl;  
    PlaceSetting(int i) {  
        super(i + 1);  
        sp = new Spoon(i + 2);  
        frk = new Fork(i + 3);  
        kn = new Knife(i + 4);  
        pl = new DinnerPlate(i + 5);  
        System.out.println("PlaceSetting constructor");  
    }  
    public static void main(String[ ] args) {  
        PlaceSetting x = new PlaceSetting(9);  
    }  
}
```



Результат?

```
Custom constructor  
Utensil constructor  
Spoon constructor  
Utensil constructor  
Fork constructor  
Utensil constructor  
Knife constructor  
Plate constructor  
DinnerPlate constructor  
PlaceSetting constructor  
  
Process finished with exit code 0
```

Пример

```
public class PlaceSetting extends Custom {
    Spoon sp = new Spoon(2);      Fork frk = new Fork(3);
    Knife kn = new Knife(4);      DinnerPlate pl = new DinnerPlate(5);
    PlaceSetting(int i) {
        super(i + 1);
        //sp = new Spoon(i + 2);
        //frk = new Fork(i + 3);
        //kn = new Knife(i + 4);
        //pl = new DinnerPlate(i + 5);
        System.out.println("PlaceSetting constructor");
    }
    public static void main(String[ ] args) {
        PlaceSetting x = new PlaceSetting(9);
    }
}
```

```
Custom constructor
Utensil constructor
Spoon constructor
Utensil constructor
Fork constructor
Utensil constructor
Knife constructor
Plate constructor
DinnerPlate constructor
PlaceSetting constructor

Process finished with exit code 0
```

Пример

```
public class PlaceSetting extends Custom {  
  
    PlaceSetting(int i) {  
        super(i + 1);  
        //sp = new Spoon(i + 2);  
        //frk = new Fork(i + 3);  
        //kn = new Knife(i + 4);  
        //pl = new DinnerPlate(i + 5);  
        System.out.println("PlaceSetting constructor");  
    }  
    Spoon sp = new Spoon(2);      Fork frk = new Fork(3);  
    Knife kn = new Knife(4);      DinnerPlate pl = new DinnerPlate(5);  
    public static void main(String[ ] args) {  
        PlaceSetting x = new PlaceSetting(9);  
    }  
}
```

```
Custom constructor  
Utensil constructor  
Spoon constructor  
Utensil constructor  
Fork constructor  
Utensil constructor  
Knife constructor  
Plate constructor  
DinnerPlate constructor  
PlaceSetting constructor  
  
Process finished with exit code 0
```

Композиция вместо наследяване

- » Както композицията, така също наследяването позволяват **поставяне подобекти в нов клас.**
- » Въпроси:
 - > Разлика между двета подхода?
 - > Кога кой подход е предпочтителен?

Композиция вместо наследяване

- » Обикновено **композицията** се използва когато искаме да включим възможностите на съществуващ клас в новия клас, но не и неговия интерфейс.
 - > Вграждаме обект, който можем да използваме за имплементиране функционалност в новия клас.
 - > Потребителят на новия клас вижда **само** интерфейса на новия клас (не интерфейса на вградения обект).
 - > За постигане на този ефект в новия клас се вграждат **private** обекти на съществуващи класове.

Композиция вместо наследяване

- » В определени случаи има смисъл да позволим на потребителя на новия клас директен достъп до композицията.
- » Т.е. да направим член-обектите **public**.
- » Когато потребителят знае, че сме обединили няколко части в едно, това улеснява разбирането на интерфейса на новия клас.

Пример

```
class Engine {  
    public void start() { }  
    public void rev() { }  
    public void stop() { }  
}  
class Wheel {  
    public void inflate (int psi) { }  
}  
class Window {  
    public void rollup() { }  
    public void rolldown() { }  
}  
class Door {  
    public Window window = new Window();  
    public void open() { }  
    public void close() { }  
}
```

Пример

```
class Car {  
    public Engine engine = new Engine() ;  
    public Wheel[] wheel = new Wheel[4] ;  
    public Door left = new Door(),  
            right = new Door(); // с 2 врати  
    public Car() {  
        for(int i = 0; i < 4; i++)  
            wheel[i] = new Wheel();  
    }  
    public static void main(String[] args) {  
        Car car = new Car();  
        car.left.window.rollup();  
        car.wheel[0].inflate(72);  
    }  
}
```

Композиция и наследяване

- » Когато извършваме наследяване създаваме специална версия на базовия клас.
 - > Това означава, че специализираме клас с общо предназначения за някаква конкретна необходимост.
 - > В примера, не е необходимо наследяване.
- » Зависимостта „**e**“ се изразява посредством наследяване.
- » Зависимостта „**има**“ се изразява посредством композиция.

Приятелски за наследени

- » Ключовата дума **protected** е свързана с наследяването.
- » В определени случаи искаме да направим нещо скрито за външния свят, но същевременно да позволяваме **достъп** за членовете на производни класове.
- » Най-доброто решение е член-данните да остават **private**
 - > Винаги трябва да запазим правото си за промяна на базовата имплементация.
- » Можем обаче, да позволим **контролиран достъп** до наследниците на базовия клас чрез **protected** методи.

Пример

```
import java.util.*;  
  
class Villian {  
    private int i;  
    protected int read() { return i; }  
    protected void set(int ii) { i = ii; }  
    public Villian(int ii) { i = ii; }  
    public int value(int m) { return m*i; }  
}  
  
public class Orc extends Villian {  
    private int j;  
    public Orc(int jj) { super(jj); j = jj; }  
    public void change(int x) { set(x); }  
}
```

change() има достъп до set(),
понеже е protected.

Поетапно нарастване на разработка



46

Постепенно нарастваща разработка

- » Разработването на програми е **постепенен** процес
 - > Независимо от предхождащите анализи когато започваме един проект нямаме отговор на всички въпроси.
 - > Изгледите за успех са по-големи, когато проектът „нараства“ еволюционно, вместо конструиране изведнъж.
- » Едно от предимствата на **наследяването** е поддръжка на постепенно нарастване на разработките.
- » Позволява включване на нов код, без това да води до грешки във вече съществуващ код.
 - > Също изолира нови грешки във вече съществуващ код.

Постепенно нарастваща разработка

- » Класовете са **ясно** разделени:
 - > Не се нуждаем от изходния код на методите за да ги използваме.
- » Въпреки, че техниката на наследяване може да бъде полезна при извършване на експерименти, след като разработката се стабилизира е необходимо преразглеждане на йерархията от класове
 - > Целта е **оптимизиране** на архитектурата.
 - > Основен замисъл на наследяването „Този нов клас е тип на този стар клас“.
 - > Програмата трябва да се занимава със създаване и обработка на обекти от различни типове, за да изрази даден модел от гледна точка на приложната област на задачата.

Преобразуване нагоре

Преобразуване нагоре

- » Най-важният аспект на наследяването не е доставянето на методи за новия клас.
- » Съществена е **зависимостта** между новия и базовия клас
 - > Обобщена като: „Новият клас е **тип** на съществуващ клас“.
 - > Не е начин за обясняване на наследяването, а се поддържа **директно** от езика.
 - > Всеки обект от производния клас е също обект от базовия клас.

Преобразуване нагоре

- » Наследяването означава, че всички методи на базовия клас са достъпни и в производния клас.
- » Всяко съобщение, което можем да изпратим към базовия клас, може да бъде изпратено и към производния.

Пример

```
import java.util.*;  
  
class Instrument {  
    public void play() { }  
    static void tune(Instrument i) {  
        // ...  
        i.play();  
    }  
}  
  
class Wind extends Instrument {  
    public static void main(String[] args) {  
        Wind flute = new Wind();  
        Instrument.tune(flute);  
    }  
}
```



Коментар?

Обектите на **Wind** са
също инструменти - имат
същия интерфейс.

Автоматично преобразуване
нагоре (upcast).

Пример

```
import java.util.*;  
  
class Instrument {  
    public void play() {}  
    static void tune(Instrument i) {  
        // ...  
        i.play();  
    }  
}  
  
class Wind extends Instrument {  
    public static void main(String[] args) {  
        Wind flute = new Wind();  
        Instrument.tune(flute);  
    }  
}
```



Коментар?

Независимо от това, че методът **tune()** е дефиниран за обекти от клас **Instrument**, той **може да се извика** с референция към обект от тип **Wind**.

Пример

```
import java.util.*;  
  
class Instrument {  
    public void play() {}  
    static void tune(Instrument i) {  
        // ...  
        i.play();  
    }  
}  
  
class Wind extends Instrument {  
    public static void main(String[] args) {  
        Wind flute = new Wind();  
        Instrument.tune(flute);  
    }  
}
```



Коментар?

Понеже Java е взискателен по
отношение на проверка на
типовете, на пръв поглед това
изглежда **необичайно**.

Пример

```
import java.util.*;  
  
class Instrument {  
    public void play() {}  
    static void tune(Instrument i) {  
        // ...  
        i.play();  
    }  
}  
  
class Wind extends Instrument {  
    public static void main(String[] args) {  
        Wind flute = new Wind();  
        Instrument.tune(flute);  
    }  
}
```



Коментар?

Но, посредством **наследяването** обект от класа **Wind** по същество представлява **също** обект от класа **Instrument**.

Пример

```
import java.util.*;  
  
class Instrument {  
    public void play() {}  
    static void tune(Instrument i) {  
        // ...  
        i.play();  
    }  
}  
  
class Wind extends Instrument {  
    public static void main(String[] args) {  
        Wind flute = new Wind();  
        Instrument.tune(flute);  
    }  
}
```



Коментар?

Преобразуване на референция
- от клас Wind в референция
към клас Instrument.

Защо преобразуване нагоре?



Проблем ю ли е преобразуването нагоре?

Не

- » Историческа причина за термина: начин, по който са били чертани диаграмите, представящи наследяването.
- » Преобразуването се извършва в посока нагоре на диаграмата на наследяване.
- » Често се нарича „преобразуване нагоре“.

Instrument

Wind

Защо преобразуване нагоре?



Зашо не е проблемно?

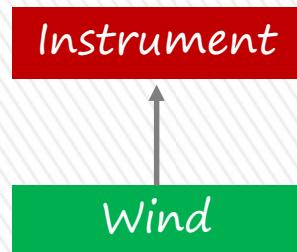


- Преминаваме от по-специфичен към по-общ тип.
- Производният клас е супер-множество на базовия клас, т.е. може да съдържа повече методи от базовия клас (минимум тях).

Защо преобразуване нагоре?



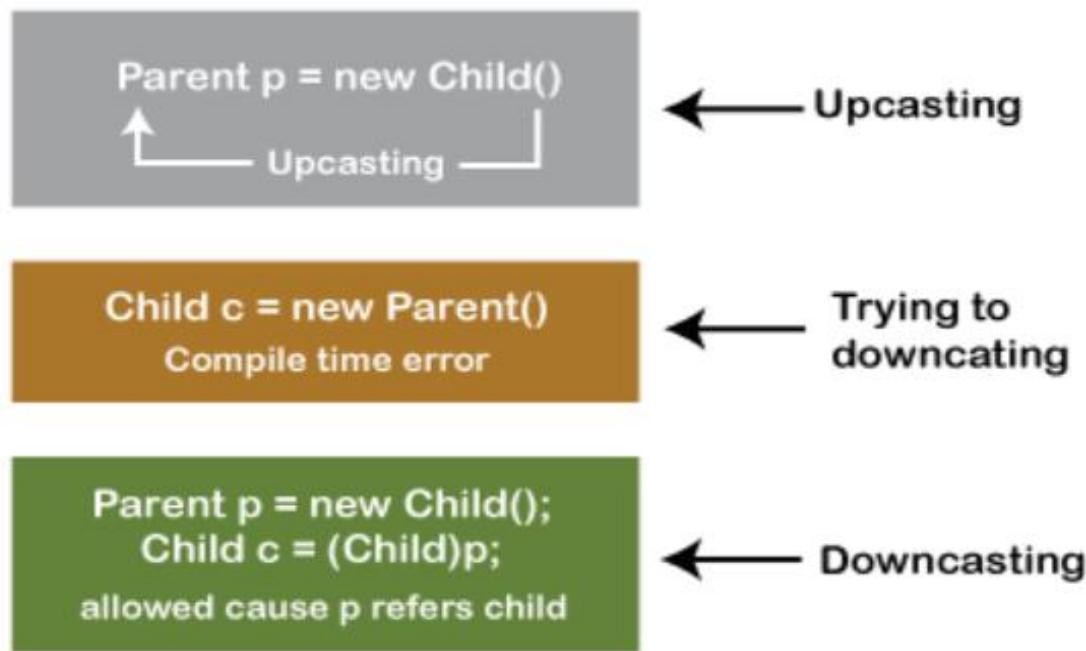
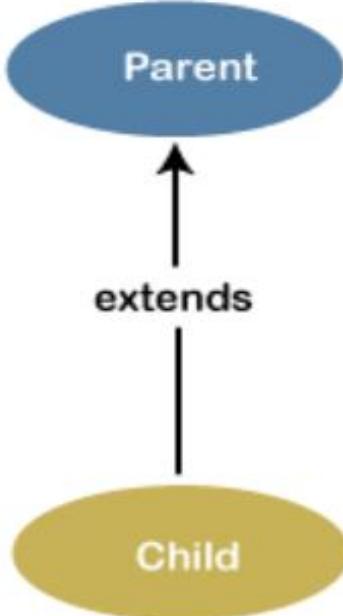
Зашто е проблемно?



- Единственото нещо, което може да се случи на интерфейса на класа е загуба, а не придобиване на методи.
- Компилаторът позволява преобразуване нагоре без никакви явни преобразования или друга специална нотация.

Възможно е също преобразуване надолу
(downcast).

Преобразуване надолу



Композиция, наследяване (обобщение)

- » В ООП:
 - > Създаваме и използваме код просто като пакетираме данни и методи заедно в един клас и използваме неговите обекти.
 - > Също, използваме вече съществуващи обекти за създаване нови посредством композиция.
 - > Въпреки, че отдаваме голямо значение на наследяването, то се използва по-рядко.
- » Композиция или наследяване?
 - > При необходимост от преобразуване нагоре: **наследяване**.

Final



62

Използване на final

- » В зависимост от контекста, ключовата дума **final** има в Java различни значения.
 - > По принцип указва елементи, които **не могат да бъдат променяни**.
- » Предотвратяване извършването на промени поради две причини:
 - > Дизайн;
 - > Ефективност.
- » Двете причини значително се различават
 - > Поради което **final** може да се използва неправилно.

Final данни

- » Указание за компилатора, че част от данните са **постоянни** (константи).
- » Константите могат да бъдат:
 - > Константи-примитиви (константи по време на компилиране);
 - > Константи-референции (константи по време на изпълнение).

Final данни

- » С константите по време на компилиране могат да се правят изчисления от компилатора, като така се подобрява времето за изпълнение.
 - > Това могат да бъдат примитиви, които се представят с ключовата дума `final`;
 - > При дефиниране на такива константи трябва да се зададе стойност;
 - > За поле, което е едновременно `static` и `final` има определено място в паметта и то не може да се променя.

Final референции

- » Константи по време на изпълнение са **референции**, които се **инициализират** по време на изпълнение:
 - > final за референции (не за примитиви);
 - > Правят референциите константи:
 - + След като референцията бъде инициализирана, **не може да бъде променяна** да сочи към друг обект.
 - + Самият обект обаче, **може да бъде променян**.
 - + Java не доставя възможност произволен обект да стане константа.

Пример

```
class Value {  
    int i = 1;  
}
```



Коментар?

```
public class FinalData {  
    final int i1 = 9;  
    static final int VAL_TWO = 99; ← константа на клас  
    public static final int VAL_THREE = 39;  
    final int i4 = (int) (Math.random()*20); ← не са константи по време  
    static final int i5 = (int) (Math.random()*20); ← на компилиране
```

```
Value v1 = new Value();  
final Value v2 = new Value();  
static final Value v3 = new Value();
```

```
final int[ ] a = { 1, 2, 3, 4, 5, 6 };
```

```
public void print(String id) {  
    System.out.println(id + ": " +  
        "i4 = " + i4 + ", i5 = " + i5);  
}
```

i1, VAL_TWO: final примитиви със стойности по време на компилиране.

Пример

```
class Value {  
    int i = 1;  
}  
  
public class FinalData {  
    final int i1 = 9;  
    static final int VAL_TWO = 99; ← константа на клас  
    public static final int VAL_THREE = 39;  
    final int i4 = (int) (Math.random()*20); ← не са константи по време на компилиране  
    static final int i5 = (int) (Math.random()*20);  
  
    Value v1 = new Value();  
    final Value v2 = new Value();  
    static final Value v3 = new Value();  
  
    final int[ ] a = { 1, 2, 3, 4, 5, 6 };  
  
    public void print(String id) {  
        System.out.println(id + ": " +  
            "i4 = " + i4 + ", i5 = " + i5);  
    }  
}
```

Коментар?

константа по време на компилиране

константа на клас

не са константи по време на компилиране

VAL_TREE: типичен начин за дефиниране на такива константи – static (само една), final (константа)

Пример

```
class Value {  
    int i = 1;  
}
```



Коментар?

```
public class FinalData {  
    final int i1 = 9;  
    static final int VAL_TWO = 99; ← константа на клас  
    public static final int VAL_THREE = 39;  
    final int i4 = (int) (Math.random()*20); ← не са константи по време  
    static final int i5 = (int) (Math.random()*20); ← на компилиране
```

```
Value v1 = new Value();  
final Value v2 = new Value();  
static final Value v3 = new Value();  
  
final int[ ] a = { 1, 2, 3, 4, 5, 6 };
```

```
public void print(String id) {  
    System.out.println(id + ": " +  
        "i4 = " + i4 + ", i5 = " + i5);  
}
```

Такива константи се именуват с главни букви, думите разделени с подчертаващо тире.

Пример

```
class Value {  
    int i = 1;  
}  
  
public class FinalData {  
    final int i1 = 9;  
    static final int VAL_TWO = 99; ← константа на клас  
    public static final int VAL_THREE = 39;  
    final int i4 = (int) (Math.random()*20); ← не са константи по време на компилиране  
    static final int i5 = (int) (Math.random()*20);  
}
```



Коментар?

```
Value v1 = new Value();  
final Value v2 = new Value();  
static final Value v3 = new Value();  
  
final int[ ] a = { 1, 2, 3, 4, 5, 6 };  
  
public void print(String id) {  
    System.out.println(id + ": " +  
        "i4 = " + i4 + ", i5 = " + i5);  
}
```

- **i5** не се променя при създаването на втори обект на класа `FinalData`.
- Стойността е `static` и се инициализира един единствен път, а не всеки път, когато се създава нов обект.

Пример

```
class Value {  
    int i = 1;  
}
```



Коментар?

```
public class FinalData {  
    final int i1 = 9;  
    static final int VAL_TWO = 99;  
    public static final int VAL_THREE = 39;  
    final int i4 = (int) (Math.random()*20);  
    static final int i5 = (int) (Math.random()*20);
```

константа по време на компилиране

константа на клас

не са константи по време
на компилиране

```
Value v1 = new Value();  
final Value v2 = new Value();  
static final Value v3 = new Value();  
  
final int[ ] a = { 1, 2, 3, 4, 5, 6 };  
  
public void print(String id) {  
    System.out.println(id + ": " +  
        "i4 = " + i4 + ", i5 = " + i5);  
}
```

- **i4, i5:** само защото са обявени за **final** не означава, че стойността им е позната по време на **компиляция**.
- Разликата между двете константи: **не-static** и **static**.

Пример

```
class Value {  
    int i = 1;  
}
```



Коментар?

константа по време на компилиране

```
public class FinalData {  
    final int i1 = 9;  
    static final int VAL_TWO = 99; ← константа на клас  
    public static final int VAL_THREE = 39;  
    final int i4 = (int) (Math.random()*20); ← не са константи по време  
    static final int i5 = (int) (Math.random()*20); ← на компилиране
```

```
Value v1 = new Value();  
final Value v2 = new Value();  
static final Value v3 = new Value();  
  
final int[ ] a = { 1, 2, 3, 4, 5, 6 };  
  
public void print(String id) {  
    System.out.println(id + ": " +  
        "i4 = " + i4 + ", i5 = " + i5);  
}
```

v1, v2, v3: демонстрират
значението на final
референциите.

Пример



Коментар?

```
public static void main(String[ ] args) {  
    FinalData fd1 = new FinalData();  
    //! fd1.i1++; // Грешка: не може да се променя стойността  
    fd1.v2.i++; // Обектът не е константен  
    fd1.v1 = new Value(); // OK - не е final  
    for(int i = 0; i < fd1.a.length; i++)  
        fd1.a[i]++; // обектът не е константен  
    //! fd1.v2 = new Value(); // Грешка: не може  
    //! fd1.v3 = new Value(); // да се променя референцията  
    //! fd1.a = new int[3];  
  
    fd1.print("fd1");  
    System.out.println("Creating new FinalData");  
    FinalData fd2 = new FinalData();  
    fd1.print("fd1");  
    fd2.print("fd2");  
}  
}
```

v2: само понеже е final не означава, че не може да се променя реферираният от нея обект.

Пример



Коментар?

```
public static void main(String[ ] args) {  
    FinalData fd1 = new FinalData();  
    //! fd1.i1++; // Грешка: не може да се променя стойността  
    fd1.v2.i++; // Обектът не е константен  
    fd1.v1 = new Value(); // OK - не е final  
    for(int i = 0; i < fd1.a.length; i++)  
        fd1.a[i]++; // обектът не е константен  
    //! fd1.v2 = new Value(); // Грешка: не може  
    //! fd1.v3 = new Value(); // да се променя референцията  
    //! fd1.a = new int[3];  
  
    fd1.print("fd1");  
    System.out.println("Creating new FinalData");  
    FinalData fd2 = new FinalData();  
    fd1.print("fd1");  
    fd2.print("fd2");  
}  
}
```

v2, v3: обаче, не могат да се обвържат повторно с нови обекти, понеже са final – това означава final за референции.

Пример



Коментар?

```
public static void main(String[ ] args) {  
    FinalData fd1 = new FinalData();  
    //! fd1.i1++; // Грешка: не може да се променя стойността  
    fd1.v2.i++; // Обектът не е константен  
    fd1.v1 = new Value(); // OK - не е final  
    for(int i = 0; i < fd1.a.length; i++)  
        fd1.a[i]++; // обектът не е константен  
    //! fd1.v2 = new Value(); // Грешка: не може  
    //! fd1.v3 = new Value(); // да се променя референцията  
    //! fd1.a = new int[3];  
  
    fd1.print("fd1");  
    System.out.println("Creating new FinalData");  
    FinalData fd2 = new FinalData();  
    fd1.print("fd1");  
    fd2.print("fd2");  
  
}  
}
```

Масивите се представят също
чрез референции.

Пример



Резултат?

```
public static void main(String[ ] args) {
    FinalData fd1 = new FinalData();
    //! fd1.i1++; // Грешка: не може да се променя стойността
    fd1.v2.i++; // Обектът не е константен
    fd1.v1 = new Value(); // OK - не е final
    for(int i = 0; i < fd1.a.length; i++)
        fd1.a[i]++; // обектът не е константен
    //! fd1.v2 = new Value(); // Грешка: не може
    //! fd1.v3 = new Value(); // да се променя референцията
    //! fd1.a = new int[3];

    fd1.print("fd1");
    System.out.println("Creating new FinalData");
    FinalData fd2 = new FinalData();
    fd1.print("fd1");
    fd2.print("fd2");

}
```

```
fd1: i4 = 7, i5 = 5
Creating new FinalData
fd1: i4 = 7, i5 = 5
fd2: i4 = 16, i5 = 5

Process finished with exit code 0
```

Final аргументи

- » Java позволява да се използват **final аргументи**.
- » Декларираме ги като такива в списъка с аргументи.
- » Това означава, че вътре в метода **не може да се променя** това, към което сочи референцията на аргументите.

Пример

```
class Gizmo {  
    public void spin() {}  
}  
  
public class FinalArguments {  
    void with(final Gizmo g) {  
        //! g = new Gizmo(); // забранено действие - g е final  
    }  
    void without(Gizmo g) {  
        g = new Gizmo(); // OK - g не е final  
        g.spin();  
    }  
    // void f(final int i) { i++; } // не може да се промени  
    // един final примитив може само да се чете  
    int g(final int i) { return i + 1; }  
  
    public static void main(String[ ] args) {  
        FinalArguments bf = new FinalArguments();  
        bf.without(null);  
        bf.with(null);  
    }  
}
```

Можем да четем аргументите, но
не можем да ги променяме.

Final методи

- » Две причини за тяхното използване.
 - > „**Заключваме**“ метода за да предотвратим евентуална промяна от наследен клас.
 - > Ефективност - компилаторът прави всички извиквания към този метод вмъкнати (*inline*).
- » **final** и **private**:
 - > Всички **private** методи в един клас са **final**.
 - > Понеже нямаме достъп до **private** методите, **не можем** да ги предефинираме.
 - > Можем да добавим спецификатора **final** към **private** - няма да предаде никакво допълнително значение.

Пример

```
class WithFinals {  
    // Ефектът е същият все едно е използвана final  
    private final void f() {  
        System.out.println("WithFinals.f()");  
    }  
    // Автоматично се приема за final  
    private void g() {  
        System.out.println("WithFinals.g()");  
    }  
}  
  
class OverridingPrivate extends WithFinals {  
    private final void f() {  
        System.out.println("OverridingPrivate.f()");  
    }  
    private void g() {  
        System.out.println("OverridingPrivate.g()");  
    }  
}
```

Пример

```
class OverridingPrivate2 extends OverridingPrivate {
    public final void f() {
        System.out.println("OverridingPrivate2.f()");
    }
    public void g() {
        System.out.println("OverridingPrivate2.g()");
    }
}
public class FinalOverridingIllusion {
    public static void main(String[ ] args) {
        OverridingPrivate2 op2 = new OverridingPrivate2();
        op2.f();
        op2.g();
        // Можем да извършим преобразувание нагоре
        OverridingPrivate op = op2;
        // но не можем да извикаме следните методи:
        //! op.f();
        //! op.g();
        // same here
        WithFinals wf = op2;
        //! wf.f();
        //! wf.g();
    }
}
```

OverridingPrivate2.f()

OverridingPrivate2.g()

Process finished with exit code 0

Final класове

- » Заявяваме, че **не искаме** да се наследява от този клас.
 - > Поради някаква причина структурата на класа е такава, че никога не възниква необходимост за извършване на промени.
 - > Също поради причини, свързани с безопасност или сигурност, не искаме да създаваме подклассове.
 - > Също по причини, свързани с по-голяма ефективност - всяко действие, свързано с този клас е колкото се може по-ефективно.

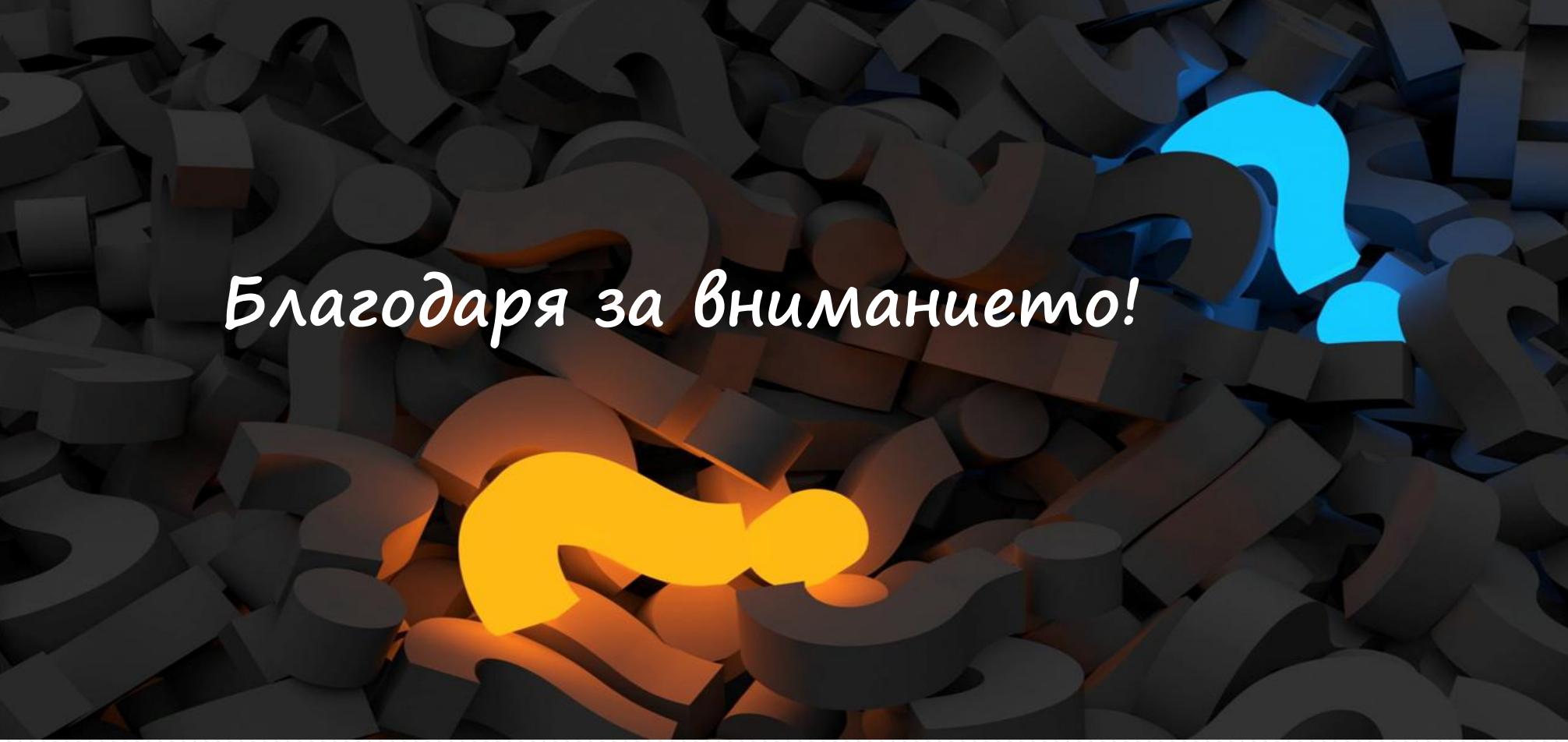
Пример

```
class SmallBrain { }

final class Dinosaur {
    int i = 7;
    int j = 1;
    SmallBrain x = new SmallBrain();
    void f() { }
}

//! class Further extends Dinosaur { }
// Грешка: не можем да наследим final class Dinosaur

public class Jurassic {
    public static void main(String[ ] args) {
        Dinosaur n = new Dinosaur();
        n.f();
        n.i = 40;
        n.j++;
    }
}
```



Благодаря за внимание!